1. Introduction
2. Modelling

In this chapter ...

Learn about:

- Kripke structures
  - formal model of a system
  - used for defining $M \models \varphi$
  - model checking algorithms operate on Kripke structures (and other models)
- Promela (language of SPIN)

Models written in higher-level languages (e.g. Promela) can be translated to Kripke structures

---

Kripke structure (1)

Kripke structures model

- states of a system $\approx$ valuation of variables + program counters
  (snapshot at some moment during execution)
- transitions: state changes

runs/computations of a system:
infinite sequences of states

atomic propositions: assertions / predicates on states
e.g. $turn = 0$, at_NC_1

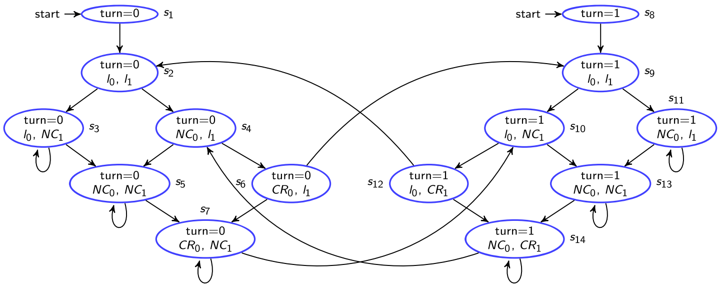
Kripke structure (2)

Definition

AP: set of atomic propositions. A Kripke structure $M = (S, S_0, R, L)$ over AP consists of

1. a set S of states,
2. a set $S_0 \subseteq S$ of initial states,
3. a transition relation $R \subseteq S \times S$;
   R is assumed to be total, i.e.:
   $\forall s \in S \, \exists s' \in S : R(s, s')$,
4. a labelling function $L : S \to 2^{AP}$;
   L gives the set of propositions which hold in a state

Our example



$S = \{s_1, \ldots, s_{14}\},\ S_0 = \{s_1, s_8\}$
$(s_1, s_2) \in R, (s_{12}, s_2) \in R, \ldots$
$L : s_2 \mapsto \{turn = 0, at\_l_0, at\_l_1\}, \ldots$

Path

Definition

A path of a Kripke structure M starting at a state s is an infinite sequence $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

Example:

$\pi_1 = s_1 s_2 s_3 s_5 s_7 s_{10} s_{12} s_2 s_3 s_5 \ldots$ path from s_1

$\pi_2 = s_3 s_3 s_3 s_3 \ldots$ path from s_3

Modelling

Models either given as

- Kripke structures
  (low-level specification) or
- in higher-level languages

models written in high-level languages are translated to Kripke structures

---

Promela and Spin

Promela (PROcess MEta LAnguage)

- (C-like) modelling language used to describe concurrent systems, e.g.
  - telecommunication protocols
  - multithreaded programs that communicate via
    - shared variables
    - message passing

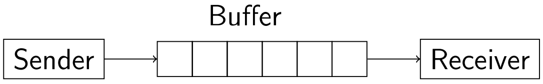
Spin (Simple Promela INterpreter)

- analysis of Promela programs
- Gerard Holzmann, 1970s, Bell Labs

Promela

Communication between processes:

- asychronous



- synchronous
  sender and receiver wait until both are ready for the communication (rendezvous, handshake)
  special case of asynchronous: length of buffer 0
- shared variables

Promela (2)

A Promela program consists of

- type declarations
- variable declarations
- channel declarations
- process declarations
- an init process

Variables and channels are *global* or *local* to processes.

Promela example

Promela program for MUTEX

```
bit turn;                        /* global variable */
proctype A0()
{
   NC_0: do                     /* busy waiting */
         :: turn == 0 -> break
         :: else -> skip
         od;

   CR_0: turn = 1;
         goto NC_0

 }
```

Process A1 similar (exchange 0 and 1)

Variables and Types

Types:

- Basic types for integers/boolean: bit (1), bool(1), byte(8), short(16), int(32)
  `bool flag;   bit turn;   short s;`
- Arrays
  `bool req[2];   bit flags[4];`
  indices starting at 0
- Records
  `typedef Record { short f; byte g; }`
  dot notation for fields: `Record r; r.f = ...;`
- Constants
  `#define N 4,   #define free (in < out)`

Processes

Declaration:

```
proctype <name> (<parameters>)
{
     < body>
}
```

body defines the behaviour of the process

this declaration defines a process, but does not execute it

Execution of processes

Two options:

1. call it from init:
   `init { ... run <name>(<actual para>); ...}`
2. declare it active
   `active proctype A() { ... }`
   if A has formal parameters, they are all initialised to 0

Example (for MUTEX):

```
init {
     run A0(); run A1();
}
```

Shared variables

Processes may share variables

```
int number = 0;

active proctype P() {
    int x = 1;
    number = number + x;
    printf("number in P = %d",number);
}

active proctype Q() {
    number = number * 2;
    printf("number in Q = %d",number);
}
```

Value of number at end?

Communication (I)

Channel declarations:
`chan <name>=[<len>] of {<type1>, ...,<typen>}`

e.g.
`chan qname = [16] of {short}` (asynchronous)
`chan port = [0] of {byte}` (synchronous)

Enumerations for defining types of messages

```
mtype = {ack,err,accept}
chan AtoB = [2] of {mtype,byte}
```

Communication(II)

Example:

```
chan c = [1] of {int,int}
```

c!x,y	values of x and y send to channel c
c?u,v	values of channel received and put into u and v
c?u,4	restricts second value: only receive if it is 4

Full/Empty channels

Channels: FIFO

if channel is empty receiver has to wait

if channel is full
- sender has to wait
or
- message is lost

(option of Spin:
full queue blocks new msgs/loses new msgs)

Example (1)

```
chan c = [0] of {int};

proctype A()
{
    int x = 0;
    c!x
}
proctype B()
{
    int y = 1;
    c?y
}
init {
   run A(); run B()
}
```

Example (2)

```
 chan b = [0] of {bit};
 chan c = [0] of {bit};

 proctype A () {
     bit x;
     b!true;
     c?x
 }
proctype B () {
     bit y;
     c!false;
     b?y;
 }
 init {
     run A(); run B();
```

Control structures (I)

Sequential composition:
`;` or `->`

Labels and jumps:

```
loop:   turn = 1;
        ...
        goto loop;
```

Empty statement:
`skip`

Control structures (II)

Branching:

```
if
:: B1 -> S1
....
:: Bn -> Sn
:: else -> Sn+1
fi
```

nondeterministic choice of a statement S_i for which the guard B_i holds

no B_i true: else

no else: wait

Control structures (III)

Iteration:

```
do
:: B1 -> S1
....
:: Bn -> Sn
:: else -> Sn+1
od
```

similar to if, but repeated after a branch has been taken

`break` exits the loop

A semaphor

```
#define p 0
#define v 1
chan sema = [0] of {bit};

proctype semaphore() {
  byte count = 1;
  end_sema: do
            :: (count == 1) -> sema!p; count = 0
            :: (count == 0) -> sema?v; count = 1
            od
}
proctype user() {
  end_user: do
            :: sema?p;  /* critical region */
            :: sema!v;  /* noncritical region */
            od
}
init { run semaphore(); run user() }
```

Some useful functions

Functions on channels

- `nempty(ch)`
  tests whether the channel `ch` is non-empty
- `empty(ch)`
  tests whether the channel `ch` is empty
- `nfull(ch)`
  tests whether the channel `ch` is not full
- `full(ch)`
  tests whether the channel `ch` is full

More on Promela at
`http://spinroot.com/spin/Man/Manual.html`

Semantics (I)

Informally:
a Promela program corresponds to a Kripke structure

- states: values of variables + program counters + contents of channels
- transitions: state changes induced by execution of statements
- initial state: determined by `init` + default initialisations of variables
- labelling with atomic propositions according to states

Semantics (II)

Kripke structure has to be finite for verification, thus

- no dynamic data structures (lists etc.)
- no unbounded channels
- no unbounded processes
- no unbounded (e.g. recursive) process creation

Semantics (III)

Interleaving semantics:

- Promela processes execute concurrently
- Non-deterministic scheduling of processes
- Processes are interleaved, i.e. statements of different processes do not occur at the same time (except handshake communication)
- all statements are atomic

Use of SPIN

Spin consists of the program `spin` itself and GUI `xspin/ispin`
Components:

- Syntax check
- Simulation
- Verification

Simulation:
- random (non-deterministic choice of next statement)
- guided (along counter example generated by verifier)
- interactive (next step chosen by user)

`http://spinroot.com/spin/Man/GettingStarted.htm`

Learned

A language for modelling reactive systems: Promela

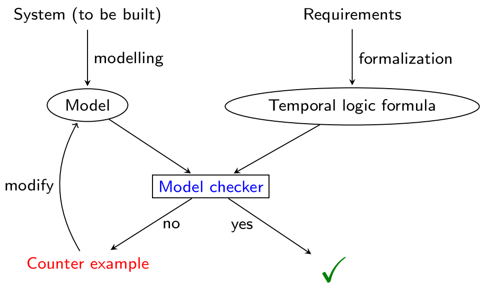
A semantic model for the language: Kripke structure

Learned

A language for modelling reactive systems: Promela

A semantic model for the language: Kripke structure

The "big picture" again:



System (to be built)

Requirements

modelling

formalization

Model

Temporal logic formula

modify

Model checker

no

yes

Counter example

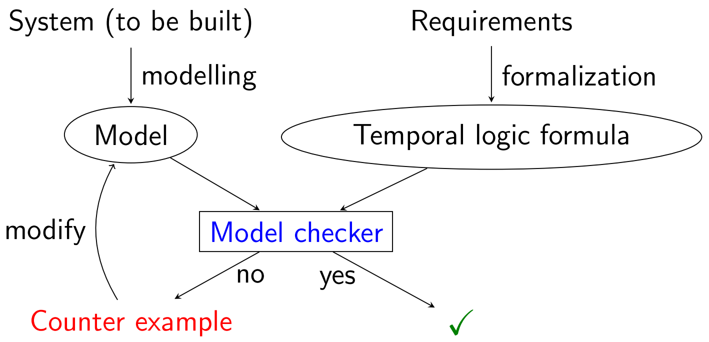
✓

Part II

LTL Model Checking

3. LTL - Syntax and Semantics
4. Verification with SPIN
5. Automata-Based LTL Model Checking

The big picture



System (to be built)

modelling

Model

Requirements

formalization

Temporal logic formula

Model checker

no

yes

Counter example

modify

✓

In this chapter ...

Learn about:

- a temporal logic (LTL)
  syntax
- semantics of LTL
  interpretation on Kripke structures
  what does $M \models \varphi$ mean ?
- examples for the specification of requirements in LTL
- expressiveness of LTL

LTL

(P)LTL - Propositional linear-time temporal logic

Basis:
  atomic propositions
  (assertions/predicates on states,
   also called state formulae)

additionally:
  boolean connectives: $\vee, \wedge, \neg$
  temporal operators: always, sometimes, tomorrow

LTL - Syntax

Definition

AP a set of atomic propositions. The set of LTL-formulae over AP is inductively defined as follows

- $p \in AP$ is an LTL formula,
- if φ is an LTL formula, so is $\neg\varphi$,
- if φ, ψ are LTL formulae, so is $\varphi \vee \psi$,
- if φ is an LTL formula, so are $X\ \varphi, G\ \varphi, F\ \varphi$,
- if φ, ψ are LTL formulae, so is $\varphi\ U\ \psi$.

A formula without $U, G\ , X\ , F$ is a state formula.

In addition ..

Derived boolean connectives

$$\begin{aligned} true &:= p \vee \neg p \\ false &:= \neg true \\ \varphi \wedge \psi &:= \neg(\neg\varphi \vee \neg\psi) \\ \varphi \Rightarrow \psi &:= \neg\varphi \vee \psi \\ \varphi \Leftrightarrow \psi &:= (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi) \end{aligned}$$

Semantics: informally

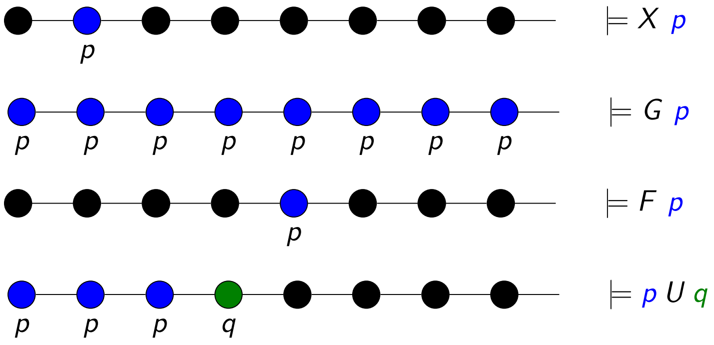
Meaning of temporal operators

- X (next)
  $X\ \varphi$: φ holds in the next state
- G (globally, always)
  $G\ \varphi$: φ holds always
- F (eventually, finally)
  $F\ \varphi$: φ holds sometimes in the future
- U (until)
  $\varphi\ U\ \psi$: φ holds until ψ holds (and ψ will eventually hold)

Examples of formulae: let $AP = \{x = 1, x < 2, x \geq 3\}$
$X\ (x = 1), \neg(x < 2), (x < 2)\ U\ (x \geq 3)$,
$F\ (x < 2) \vee G\ (x \geq 3)$

Semantics: graphically

Formulae are interpreted on paths of Kripke structures



Semantics: formal

Given Kripke structure M, look at all paths of M

Definition

Let M be a Kripke structure and φ an LTL formula.
$M \models \varphi$ iff $\pi \models \varphi$ for all paths π of M, which start in the initial state.

some notation:
$\pi = s_0 s_1 s_2 \dots$ a path.
$\pi^i = s_i s_{i+1} s_{i+2} \dots$ is the i-suffix of π

Semantics: formal (2)

Definition

Let $\pi = s_0 s_1 s_2 \dots$ a path, φ an LTL formula. $\pi \models \varphi$ is inductively defined as follows.

- $\pi \models p, p \in AP$ iff p holds in s_0 (i.e. $p \in L(s_0)$),
- $\pi \models \neg\varphi$ iff not $\pi \models \varphi$,
- $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ or $\pi \models \psi$,
- $\pi \models X\ \varphi$ iff $\pi^1 \models \varphi$,
- $\pi \models G\ \varphi$ iff $\forall\, i \geq 0 : \pi^i \models \varphi$,
- $\pi \models F\ \varphi$ iff $\exists\, j \geq 0 : \pi^j \models \varphi$,
- $\pi \models \varphi\ U\ \psi$ iff $\exists\, k \geq 0 : \pi^k \models \psi$ and $\forall\, j, 0 \leq j < k, \pi^j \models \varphi$.

Examples

on the blackboard